PUC Minas
A B A K Ó S
Instituto de Ciências Exatas e Informática

Practical Work 1*

Model - Practical work TGC - ICEI - PUC Minas

Andrei Gonçalves R. Massaini and Luiz Fernando C. Rodrigues[1]

Summary

The practical work involves identifying blocks in a biconnected graph, which is a connected graph without joints, that is, a "fault-tolerant" graph. A graph is biconnected if and only if each pair of its vertices is connected by two internally disjoint paths. The biconnected components of the graph are maximal subgraphs of G that are biconnected at vertices. Each biconnected component is also called a graph block.

The practical work proposes three methods for identifying blocks in a biconnected graph: (i) a method that verifies the existence of two internally disjoint paths (or a cycle) between each pair of vertices in the block; (ii) a method that identifies joints by testing connectivity after removing each vertex; and (iii) the method proposed by Tarjan (1972).

The experiment consists of evaluating the average time spent for the three strategies applied to random graphs containing 100, 1,000, 10,000 and 100,000 vertices. **Key words:** Template. LaTEX. Abakos. Periodicals.

*Article presented to Revista Abakos

[1]Bachelor of Computer Science PUC Minas, Brazil – andrei.massaini@hotmail.com

1. INTRODUCTION

This work aims to identify blocks in a graph. Blocks are connected subgraphs that cannot be decomposed into smaller subgraphs. The identification of these blocks is important in several areas, such as social networks, analysis of electrical circuits, analysis of communities in complex networks, among others. To identify blocks, we use specific algorithms that explore the structure of the graph and its properties. In this work, we will present the implementation of these algorithms and the analysis of the results obtained in graphs of different sizes and topologies. Furthermore, we will discuss the implications of these results for practical applications and possible future research directions.

2 IMPLEMENTATION

In developing the work, we used the Java language, as it is a widely adopted and quite versatile language. Initially, we started from representing a graph through an adjacency matrix, but implementing a graph through an adjacency list was necessary, since we were unable to generate the graph with 100,000 vertices due to memory overflow.

2.1 Implemented graphs

In the 'Implemented graphs' folder, there are two classes that are used to represent tar the graphs. One through an adjacency matrix, and the other through an adjacency list. Both have the basic methods of instantiating based on the size of vertices, as well as methods of adding edges passing their respective vertices as a parameter.

2.1.1 Algorithms

The algorithms were divided into two folders: The 'ListMethods' folder stores all the methods that were implemented using the adjacency list, while the 'MatrizMethods' folder stores the methods that were implemented using the adjacency matrix.

2.1.1.1 Disjoint Paths - 1

The algorithm implements a strategy to identify whether a graph is biconnected, that is, whether each pair of vertices is connected by two internally disjoint paths. The idea

The main thing is to use the Floyd-Warshall algorithm to calculate the distances between all pairs of vertices in the graph, and then check whether there are two disjoint paths between each pair of vertices.

The Floyd-Warshall algorithm is used to calculate the shortest distances between all pairs of vertices in the graph. To do this, initially, the distance and proximity matrices are initialized with the values of the input graph. Then, the Floyd-Warshall algorithm is executed, comparing the distances already calculated with the possibility of obtaining a smaller distance passing through an intermediate vertex. If this is possible, the distance matrix is updated with the new distance and the proximity matrix is updated with the new intermediate vertex.

After executing the Floyd-Warshall algorithm, the algorithm checks whether there are two disjoint paths between each pair of vertices in the graph. To do this, the distance matrix is traversed and for each pair of vertices, it is checked whether there are two disjoint paths passing through an intermediate vertex. If they exist, the intermediate vertex is marked as a joint.

Finally, the algorithm returns a Boolean vector indicating whether each vertex is a joint. relationship or not. If a vertex is marked as joint, it means that there are at least two disjoint paths that connect it to other vertices, which indicates that the graph is biconnected.

2.1.1.2 Find Components - 2

The algorithm consists of finding the articulation points in a graph represented by an adjacency matrix. To do this, we remove each vertex from the graph, and count the number of components in it. If the number of components is greater than 1, the removed vertex is a joint. The algorithm is implemented through three functions: getNum-ConnectedComponents which counts the number of connected components using depth-first search (DFS); removeVertex which removes a vertex from the adjacency matrix and returns the new matrix; and getArticulacoes which goes through all the vertices of the adjacency matrix, removes one at a time and counts the number of connected components to identify the articulation points. The result is stored in a vector of integers, where the value 1 indicates that the corresponding vertex is a joint and 0 otherwise.

2.1.1.3 Tarjan - 3

Tarjan's algorithm aims to find the Strongly Connected Components (SCCs) in a graph. SCCs are subgraphs in which it is possible to reach from one vertex to any other vertex of that subgraph.

The code starts by defining some instance variables, such as the adjacency matrix, the size of the matrix, identifiers, low identifiers, a stack, among others. Then, it implements the "calcularSCCs" method, which starts the process of searching for SCCs.

This method calls the "visit" method, which is the core of Tarjan's algorithm.

The "visit" method visits each vertex of the graph, assigns an identifier and a low identifier to the vertex, and pushes it onto the stack. It then traverses the adjacent vertices and, if a vertex has not yet been visited, calls the "visit" method again recursively. The method also updates low identifiers based on adjacent vertices visited.

If the visited vertex is the start of an SCC, the method creates a list of SCCs and fills it with unstacked vertices from the stack up to the start vertex. This is repeated until all vertices have been visited and all SCCs have been identified.

3 RESULTS ANALYSIS

In this section the performance of the algorithms will be analyzed based on the time taken to find the blocks.

When executing the algorithm for 100, 1,000, 10,000 and 100,000 graph vertices, for each of the three proposed methods, we arrived at different execution time results for each situation, varying the number of vertices and the method used, one per matrix adjacency list and another by ajacency list.

A time stamp code was added to each method proposed above with the purpose of comparing their efficiency. When using an adjacency matrix, we were able to perform the tests only with graphs of up to 1,000 vertices. When trying to run the algorithm for graphs of 10,000 vertices or more, it was not possible to perform the measurement, as it gave TIME OUT. By using the adjacency list, we were able to carry out tests with all proposed graphs, there was no TIME OUT and the processing speed was much faster.

3.1 Experiments

The tests were carried out on a notebook with an i7 8700k OC 5.0Ghz processor, 16Gb RAM.

Algorithm execution time (ms)

Method	[V]	Adjacency Matrix	Adjacency List
i	100	8	7
ii	100	13	7
iii	100	1	1
i	1,000	1885	1136
ii	1,000	2841	119
iii	1,000	7	1
i	10,000	TIMEOUT	TIMEOUT
ii	10,000	TIMEOUT	TIMEOUT
iii	10,000	TIMEOUT	TIMEOUT
i	100,000	TIMEOUT	TIMEOUT
ii	100,000	TIMEOUT	TIMEOUT
iii	100,000	TIMEOUT	TIMEOUT

Source: Research data

4 CONCLUSION

After carrying out 90 tests with the Adjacency Matrix, 10 for each method, and 90 tests with the Adjacency List, also 10 for each method, several differences and observations can be verified.

Although both algorithms are capable of performing the three methods, it can be seen that for a small graph, with approximately 100 vertices, both are capable of performing them, however the algorithm that uses the Adjacency List has an efficiency of little better than the one that uses Adjacency Matrix only for the second method, approximately 50 percent, as for the first method and the third there was almost no difference. When we evolved the graphs to approximately 1,000 vertices, both algorithms were efficient, however the use of Adjacency List was approximately 65 percent faster in the first method, 96 percent faster in the second method and 84 percent faster in the third method.

However, we were able to see that the best way to find the blocks, through the three methods used, is by implementing an Adjacency List, and using the third proposed method, the Tarjan method.